

Ensure that the Bounds of No Memory Region Are Violated

William L. Fithen, Software Engineering Institute [vita³]

Copyright © 2005 Carnegie Mellon University

2005-10-03

L4 / D/P⁴

Violation of memory bounds can introduce vulnerability.

Description

Buffer overflows occur when data are written or read outside of the boundaries of the memory allocated to a particular data structure [Farrow 02⁷]. C and C++ are susceptible to buffer overflows because these languages

- define strings as null-terminated arrays of characters (as opposed to languages where strings explicitly carry their length around with them),
- do not perform implicit bounds checking, and
- provide standard library functions that operate on strings but which do not require enough input information to enforce bounds checking.

Depending on the location of the memory and the size of the overflow, a buffer overflow can occur without impact to normal program operations, can result in anomalous behavior or data corruption, or can lead to abnormal program termination (possibly causing a denial of service).

Buffer overflows are troublesome in that they can go undetected during the development and testing of software applications. C and C++ compilers do not always identify defects that can lead to buffer overflows during compilation or produce code that reports out-of-bounds reads or writes at runtime. Static analysis tools can, in some cases, detect code likely to allow buffer overflows and dynamic analysis tools can be used to discover buffer overflows with 100% reliability but only for cases where the test data actually precipitates an overflow [BSI:Tools⁸].

All buffer overflows are defects; not all buffer overflows lead to software vulnerabilities. However, a buffer overflow can lead to a vulnerability when an attacker can manipulate user-controlled inputs to violate an explicit or implicit security policy⁹. There are, for example, well-known techniques for overwriting frames in the stack to execute arbitrary code [Aleph One 96¹⁰]. Buffer overflows can also be exploited in heap or static memory areas by overwriting data structures in adjacent memory. A buffer overflow occurs when a data store operation extends beyond the bounds of the targeted memory region. This alters surrounding memory, creating the potential for those alterations to be influenced by an adversary.

This is the single largest—and today probably the most damaging—class of vulnerability in the CERT/CC database. In practice nearly all of these types of vulnerabilities occur in C or C++, and many of them are directly attributable to the use of functions like strcpy, strcat, sprintf, and similar functions instead of the corresponding bounded versions strncpy, strncat, and snprintf [BSI:Knowledge:Coding Practices¹¹, BSI:Knowledge:Coding Rules¹²].

3. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/320-BSI.html (Fithen, William L.)

7. <file:///Users/wlf/Workspaces/Eclipse-3.1/swa-content/documents/html-upload/knowledge/guidelines/buffer-overflow.html#Farrow-02>

8. <http://buildsecurityin.us-cert.gov/bsi/articles/tools.html> (Tools)

9. Understanding the skills of your adversaries is critical. Exactly how much does your adversary control? If your adversary controls no input, then the buffer overflow is not a vulnerability (though it remains a defect). If your adversary controls more aspects of input, then the risk of a successful exploitation of a buffer overflow increases, eventually reaching the point of practically guaranteeing a compromise.

10. #dsy323-BSI_refs

11. <http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/coding.html> (Coding Practices)

12. <http://buildsecurityin.us-cert.gov/bsi/76-BSI.html> (Coding Rules)

Classifying buffer overflow vulnerabilities is difficult because of the broad number of attacks that are considered buffer overflows. The minimum requirement for a vulnerability to be a buffer overflow vulnerability is that it is exploited by causing a vulnerable program to write outside the bounds of a data structure.

One way to classify a buffer overflow, therefore, is the location of the buffer that is overwritten. Even this is somewhat confused because of the different ways process memory can be organized, but since data usually exist in either the stack, heap, or data segments we can generally classify buffer vulnerabilities based on which of those locations contains the vulnerable buffer.

A successful exploit must overflow a buffer for a purpose. The purpose may be to modify the value of a variable, data pointer, function pointer, or return address on the stack. Modification of a variable may be used to change some behavior of a program, possibly making it vulnerable to further attack. Modification of a data pointer, function pointer, or return address can all be used to execute arbitrary code.

For the exploit to execute code, the code must already exist in the address space of the vulnerable process (presumably in the code segment) or it must be injected. Code could be injected into the stack, heap, or data segments. Where the code is injected can be relevant if one or more memory segments is made to be non-executable [Hoglund 04¹³].

- Buffer underflows are also the same class but are fairly rare and can occur only in languages that can do negative pointer arithmetic or allow negative array indices (such as C and C++).
- Buffer overflows can occur in languages besides C and C++.
 - In some dialects of FORTRAN, buffer overflows can occur when two adjacent arrays of the same type exist.
- Some older versions of TeX allocated large static arrays to use as dynamic memory pools. Buffer overflows could potentially occur within that type of array, since the language would interpret any access into the array as legal.
- Bounds violation can occur on read or write operations. The term buffer overflow refers only to the write version.
 - There have been read-type bounds violation vulnerabilities, but no one has named them; they tend to be classed as "leaks."

Applicable Context

- The implementation programming language does not enforce memory bounds.
- The program does not check its own memory bounds.
- An adversary can cause the program to read or write memory outside of those bounds.

Impacts Being Mitigated

- Impact #1:
 - **Minimally:** The minimal impact of a buffer overflow is nil. It is entirely possible for a buffer overflow to be either unexploitable or have no effect. The kind of buffer overflow that has no impact is one where corrupted memory is never subsequently accessed. In that case, the buffer overflow might never be apparent.
 - **Maximally:** The maximal impact of a buffer overflow depends on the technicality of the overflow and how it can be exploited. In every case memory is corrupted and then used in some fashion that results in some degree of incorrect program behavior. This behavior can range from introducing subtle logic errors to the execution of arbitrary adversary-supplied machine code.

13. #dsy323-BSI_refs

Security Policies to be Preserved

- Policy #1
 - Developers expect program memory to retain integrity throughout program execution.

How to Recognize this Defect

- Look for the use of functions, especially those provided by others, that accept arrays (e.g., `char *`) and do not also accept an additional parameter to provide the length of the array.
- Look for iterations over arrays to ensure that array bounds are not violated (negative or positive). Pay special attention to "fence post" errors.
- Look for conversions to and from pointers that obfuscate the target of the pointer. Such type ambiguity can lead to memory bounds violations or a number of other kinds of errors.
- [BSI:Knowledge:Coding Practices³¹]
- [BSI:Knowledge:Coding Rules³³]
- [BSI:Tools³⁴]

Mitigation Advice

To Engineers:

- **Efficacy:** HIGH
- All of the following should be observed:
 - All array boundaries should be checked.
 - All pointer-based memory accesses should be checked to ensure they access the intended memory region.
- Much has been written along these lines [Howard 05⁴³, Hoglund 04⁴⁴, Simon 01⁴⁵, Wheeler 04⁴⁶, Aleph One 96⁴⁷].

References

[Aleph-One 96]

Aleph One. "Smashing the Stack for Fun and Profit." *Phrack Magazine* 7, 49 (1996): File 14 of 16. <http://www.phrack.org/phrack/49/P49-14>.

[Farrow 02]

Farrow, Rik. *What Are Buffer Overflows?* <http://www.watchguard.com/infocenter/editorial/135136.asp> (2002).

[Hoglund 04]

Hoglund, Greg & McGraw, Gary. *Exploiting Software: How to Break Code*. Boston, MA: Addison-Wesley, 2004.

[Howard 05]

Howard, Michael; LeBlanc, David; & Viega, John. *19 Deadly Sins of Software Security*. Emeryville, CA: McGraw-Hill Osborne Media, 2005.

[Simon 01]

Simon, Istvan. *A Comparative Analysis of Methods of Defense against Buffer Overflow Attacks*. <http://www.mcs.csu Hayward.edu/~simon/security/boflo.html>⁵¹ (2001).

Carnegie Mellon Copyright

Copyright © Carnegie Mellon University 2005-2010.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu¹.

The Build Security In (BSI) portal is sponsored by the U.S. Department of Homeland Security (DHS), National Cyber Security Division. The Software Engineering Institute (SEI) develops and operates BSI. DHS funding supports the publishing of all site content.

NO WARRANTY

THIS MATERIAL OF CARNEGIE MELLON UNIVERSITY AND ITS SOFTWARE ENGINEERING INSTITUTE IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

1. <mailto:permission@sei.cmu.edu>